APPENDIX A

```
package junglee.vdbms.wdk.visual;
import java.io.*;
import java.awt.*;
import java.util.*;

/**
  * <b> WrapperModel Class </b>
  *
  * <p>This class is used to model wrappers. A wrapper is
  * identified as a set of operations, some of which are connected
  * by links. Supported functions are:
  * <ul>
  * <li> construction of wrappers by adding or deleting links and
operations. <\li>
  * <li> rendering wrappers on a canvas using the WrapperView class.
</li>
  * <li> response to user input using the WrapperController class
  * <li> shifting all links and operations. </li>
  * <li> scaling operations around center of gravity. </li>
  * </ul>
  * <p>Copyright (c) 1997, Junglee Corp.
  * @title WrapperModel
  * @author Stephan Erickson
  * @version 1.0
  *
  */

public class WrapperModel
  implements Serializable
{

  ////////////////// Constructor ////////////////

  /** Create a wrapper model, and make it the current wrapper. **/
  public WrapperModel() {
  }

  ////////////////// Methods ////////////////

  /** Get vector of wrapper listeners **/
  public Vector getWrapperListeners()
  {
    return wrapperListeners;
  }

  /** Allow a Visual GUI to register interest in events. **/
  public void addWrapperListener(WrapperListener wl)
  {
    wrapperListeners.addElement(wl);
  }

  /** Get id **/
  public String getId()
  {
    return theId;
  }
```

```
  /** Set the id **/
  public void setId(String id)
  {
    theId = id;
  }

    /** Get root operation **/
  public Operation getRoot()
  {
    return root;
  }

  /** Set the root operation **/
  public void setRoot(Operation op)
  {
    root = op;
  }

  /** Get Initialization File **/
  public String getInitFile()
  {
    return theInitFile;
  }

  /** Set Initialization File **/
  public void setInitFile(String initFile)
  {
    theInitFile = initFile;
  }

  /** Add breakpoint to operation **/
  public void addBreakpoint(Operation op)
  {
    op.isBreakpoint = true;
  }

  /** Remove breakpoint from operation **/
  public void removeBreakpoint(Operation op)
  {
    op.isBreakpoint = false;
  }

  /** Does operation have breakpoint? **/
  public boolean isBreakpoint(Operation op)
  {
    return op.isBreakpoint;
  }

  /** Warn message **/
  public static void warn(Operation from, String message) {
    System.out.println("In operation " + from.getId() + ": " +
message);
  }

  /** Warn message **/
  public static void warn(String message) {
    System.out.println(message);
  }
```

```
  /** Print message **/
  public static void print(String message) {
    System.out.println(message);
  }

   //******************* MODEL OPERATIONS ********************

  /**
    * Add operation to wrapper model
    * @param operation the operation to add
    * Does nothing if operation is null
    */

  public void addOperation(Operation operation)
  {
    if (operation!=null) {
      operation.setId(getNextOperationId());
      operation.setWrapperModel(this);
      if (!hasOperation(operation)) {
        theOperationVector.addElement(operation);
      }
    }
  }

  /**
    * Add operation to wrapper model, and group with
    * existing operation list
    * @param operation the operation to add
    * Does nothing if operation is null
    */

  public void addOperation(Operation operation, OperationList
operationList)
  {
    // Add operation to operation list
    if (operation!=null) {
      operationList.addOperation(operation);
      operation.setId(getNextOperationId());
      operation.setWrapperModel(this);
    }
    // Add operation list to wrapper
    if (operationList!=null) {
      if (operation!=null)
        operationList.setId(-operation.getId());
      operationList.setWrapperModel(this);
      if (!theOperationListVector.contains(operationList)) {
        theOperationListVector.addElement(operationList);
      }
    }
  }

  /**
    * Determine if operation is in wrapper
    * @param operation the operation
    * Returns false if operation is null
    */

  public boolean hasOperation(Operation operation)
  {
```

```
    if (operation!=null) {
      Enumeration e = theOperationVector.elements();
      for (;e.hasMoreElements();) {
        Operation v = (Operation)e.nextElement();
        if (v.getId()==operation.getId())
          return true;
      }
    }
    return false;
  }

  /**
    * Remove Operation
    * @param operation the operation to remove
    * Does nothing if operation is null
    */

  public void removeOperation(Operation operation)
  {
    if (operation!=null) {
      removeLinks(operation);
      // Remove bound operation
      OperationList operationList = getOperationList(operation);
      if (operationList!=null) {
        operationList.removeOperation(operation);
        // If operation list is empty, remove list
        if (operationList.getOperations().size()==0) {
          removeOperationList(operationList);
        }
      } else {
        // Remove free operation
        theOperationVector.removeElement(operation);
      }
    }
  }

  /**
    * retrieve operations associated with wrapper
    * @return vector of operations
    */

  public final Vector getOperations()
  {
    return theOperationVector;
  }

   /**
    * set operations associated with wrapper
    * @param vector of operations
    */

  public final void setOperations(Vector operations)
  {
    theOperationVector = operations;
  }

  /**
    * retrieve operation edge is pointing to
    * @return Operation
```

```
   */

  public final Operation getOperation(Operation operation, String
linkName)
  {
    Vector links = new Vector();
    Enumeration e = theLinkVector.elements();
    for (;e.hasMoreElements();) {
      Link link = (Link)e.nextElement();
      if (link.getStartOperation()==operation) {
        if (link.getLabel().equals(linkName))
          return link.getEndOperation();
      }
    }
    return null;
  }

  /**
    * retrieve operation associated with id
    * @return Operation or null if no operations can be found
    */

  public final Operation getOperation(int id)
  {
    Enumeration e = theOperationVector.elements();
    for (;e.hasMoreElements();) {
      Operation o = (Operation)e.nextElement();
      if (o.getId()==id) return o;
    }
    Enumeration e2 = theOperationListVector.elements();
    for (;e2.hasMoreElements();) {
      OperationList ol = (OperationList)e2.nextElement();
      if (ol.getId()==id) return ol;
    }
    return null;
  }

  /**
    * Add Link
    * @param link the link to add.
    * Does nothing if link is null.
    */

  public void addLink(Link link)
  {
    if (link!=null) {
      theLinkVector.addElement(link);
    }
  }

  /**
    * Remove Link
    * @param link the link to remove
    * Does nothing if link is null
    */

  public void removeLink(Link link)
  {
    if (link!=null) {
```

```
      theLinkVector.removeElement(link);
    }
  }

  /**
    * Remove all links connected to a given operation
    * @param operation from which links should be removed
    * Does nothing if operation is null
    */

  public void removeLinks(Operation operation)
  {
    if (operation!=null) {
      // Find links to delete
      Vector deleteVector = new Vector();
      Enumeration e = theLinkVector.elements();
      for (;e.hasMoreElements();) {
        Link link = (Link)e.nextElement();
        if (link.getStartOperation()==operation ||
            link.getEndOperation()==operation) {
          deleteVector.addElement(link);
        }
      }

      // Delete list of links
      Enumeration e2 = deleteVector.elements();
      for (;e2.hasMoreElements();) {
        Link link = (Link)e2.nextElement();
          theLinkVector.removeElement(link);
      }
    }
  }

  /**
    * retrieve links associated with wrapper
    * @return vector of links
    */

  public final Vector getLinks()
  {
    return theLinkVector;
  }

  /**
    * set links associated with wrapper
    * @param vector of links
    */

  public final void setLinks(Vector links)
  {
    theLinkVector = links;
  }

  /**
    * retrieve links starting from operation
    * @return vector of links
    */

  public final Vector getLinks(Operation operation)
```

```
    {
      Vector links = new Vector();
      Enumeration e = theLinkVector.elements();
      for (;e.hasMoreElements();) {
        Link link = (Link)e.nextElement();
        if (link.getStartOperation()==operation ||
            link.getEndOperation()==operation) {
          links.addElement(link);
        }
      }
      return links;
    }

    /**
      * Update link ids after unserialization process
      * @param vector of links
      */

    public final void updateLinkIds()
    {
      Enumeration e = theLinkVector.elements();
      for (;e.hasMoreElements();) {
        Link link = (Link)e.nextElement();

        // Check if link is valid
        String linkName = link.getLabel();
        Operation                    startOperation                    =
  getOperation(link.getStartOperationId());
        boolean foundValidLink = false;
        if (startOperation!=null) {
          String[] linkArray = startOperation.getLinkNames();
          for (int i=0; i<linkArray.length; i++) {
            String validName = linkArray[i];
            if (validName.equals(linkName)) {
              foundValidLink = true;
            }
          }
        }

        // Remove link if it is not valid
        if (!foundValidLink || startOperation==null) {
          removeLink(link);
        }

        // Update operation links
        Enumeration e2 = theOperationVector.elements();
        for (;e2.hasMoreElements();) {
          Operation operation = (Operation)e2.nextElement();
          // Update wrapper model
          operation.setWrapperModel(this);
          if (link.getStartOperationId() == operation.getId()) {
            link.setStartOperation(operation);
          }
          if (link.getEndOperationId() == operation.getId()) {
            link.setEndOperation(operation);
          }
        }
        // Update operation list links
        Enumeration e3 = theOperationListVector.elements();
```

```
      for (;e3.hasMoreElements();) {
        OperationList operationList = (OperationList)e3.nextElement();
        // Update wrapper model
        operationList.setWrapperModel(this);
        if (link.getStartOperationId() == operationList.getId()) {
          link.setStartOperation(operationList);
        }
        if (link.getEndOperationId() == operationList.getId()) {
          link.setEndOperation(operationList);
        }
        // Update operation list element links
        Vector operations = operationList.getOperations();
        Enumeration e4 = operations.elements();
        for (;e4.hasMoreElements();) {
          Operation operation = (Operation)e4.nextElement();
          // Update wrapper model
          operation.setWrapperModel(this);
          if (link.getStartOperationId() == operation.getId()) {
            link.setStartOperation(operation);
          }
          if (link.getEndOperationId() == operation.getId()) {
            link.setEndOperation(operation);
          }
        }
      }
    }
  }

  /**
    * Update link ids after unserialization process
    * @param vector of links
    */

  public final void updateRootOperation()
  {
    Operation root = getOperation(1);
    setRoot(root);
    if (root!=null)
      root.setWrapperModel(this);
  }

  /**
    * Remove operation list
    * @param operationList the operation list to remove
    * Does nothing if operation list is null
    */

  public void removeOperationList(OperationList operationList)
  {
    if (operationList!=null) {
      removeLinks(operationList);
      theOperationListVector.removeElement(operationList);
    }
  }

  /**
    * Return operation list containing operation
    * @param operation the operation
    * @return operation list containing operation or null
```

```
  */

public OperationList getOperationList(Operation operation)
{
  Enumeration e = getOperationLists().elements();
  for (;e.hasMoreElements();) {
    OperationList operationList  = (OperationList)e.nextElement();
    Enumeration e2 = operationList.getOperations().elements();
    for (;e2.hasMoreElements();) {
      Operation v = (Operation)e2.nextElement();
      if (v==operation)
        return operationList;
    }
  }
  return null;
}

/**
  * retrieve operation lists associated with wrapper
  * @return vector of links
  */

public final Vector getOperationLists()
{
  return theOperationListVector;
}

/**
  * set operation lists associated with wrapper
  * @param vector of operation lists
  */

public final void setOperationLists(Vector operationLists)
{
  theOperationListVector = operationLists;
}

/**
  * retrieve next unused operation id
  * @return operation id
  */

public int getNextOperationId()
{
  int i;
  for (i=1; i<Integer.MAX_VALUE; i++) {
    boolean cont = false;
    Enumeration e = theOperationVector.elements();
    Enumeration e2 = theOperationListVector.elements();
    for (;e.hasMoreElements();) {
      Operation operation = (Operation)e.nextElement();
      if (i==operation.getId())
        cont = true;
    }
    for (;e2.hasMoreElements();) {
      OperationList operationList = (OperationList)e2.nextElement();
      Enumeration e3 = operationList.getOperations().elements();
      for (;e3.hasMoreElements();) {
        Operation operation = (Operation)e3.nextElement();
```

```
            if (i==operation.getId())
              cont = true;
          }
        }
        if (!cont) break;
      }
      return i;
    }

    //******************* SELECTION OPERATIONS ********************

    /**
      * Set current selected wrapper element
      * @param e new selected wrapper element
      */

    public void setSelectedElement(WrapperElement e)
    {
      e.theSelectedElement = e;
    }

    /**
      * Return current selected wrapper element
      * @return wrapper Element
      */

    public WrapperElement getSelectedElement()
    {
      return WrapperElement.theSelectedElement;
    }

    /**
      * Is wrapper element selected?
      * @param e wrapper element
      * @return true/false
      */

    public boolean isSelectedElement(WrapperElement e)
    {
      return (e==WrapperElement.theSelectedElement);
    }

    //******************** DATA ********************

    // The wrapper model in current use
    private static WrapperModel theWrapper = null;

    // The wrapper listeners
    Vector wrapperListeners = new Vector();

    // The root operation
    Operation root;

    // List of operations
    private Vector theOperationVector = new Vector();

    // List of operation lists
    private Vector theOperationListVector = new Vector();
```

```
    // List of links
    private Vector theLinkVector = new Vector();

    // The id of the wrapper
    private String theId = "*thunk*";

    // The initialization file
    private String theInitFile = null;
}
```

APPENDIX B

```
package junglee.vdbms.wdk.visual;
import java.awt.*;
import java.io.*;
import com.sun.java.swing.*;

/**
  * <b> Wrapper Element Class </b>
  *
  * <p>A wrapper element is either an operation or an operation list,
or a link
  * between two operations. Wrapper elements have a label, color,
size, associated
  * wrapper model and information about whether they are selected or
not. They also
  * can have an associated image, and id.
  *
  * <p>Copyright (c) 1997, Junglee Corp.
  * @title GraphElement
  * @author Stephan Erickson
  * @version 1.0
  */

public class WrapperElement
  implements Serializable
{

  /**
    * Constructor
    *
    */

  public WrapperElement()
  {
  }

 /**
   * Set label
   * @param label set label of element
   */

  public void setLabel(String label)
  {
    theLabel = label;
    FontMetrics                                fm                    =
Toolkit.getDefaultToolkit().getFontMetrics(theFont);
    theLabelWidth = fm.stringWidth(label);
    theLabelHeight = fm.getHeight();
  }

 /**
   * Get label
   * @return get label of element
   */

  public String getLabel()
  {
    return theLabel;
```

```
  }

 /**
   * Set color
   * @param color new color of element
   */

  public void setColor(Color color)
  {
    theColor = color;
  }

 /**
   * Get color
   * @return current color of element
   */

  public Color getColor()
  {
    return theColor;
  }

 /**
   * Set selected color
   * @param color new selected color
   */

  public void setSelectedColor(Color color)
  {
    theSelectedColor = color;
  }

 /**
   * Get selected color
   * @return current selected color
   */

  public Color getSelectedColor()
  {
    return theSelectedColor;
  }

 /**
   * Set size
   * @param size set size of element
   * Do nothing if new size is below minimum allowed size.
   */

  public void setSize(int size)
  {
    //if (size>theMinimumSize)
      theSize = (double) size;
  }

 /**
   * Get size
   * @return get size of element
   */
```

```
  public int getSize()
  {
    return (int) theSize;
  }

 /**
   * Set selected
   * @param selected whether or not element is selected
   */

  public void setSelected(boolean selected)
  {
    if (selected==true)
      theSelectedElement = this;
    else
      theSelectedElement = null;
  }

 /**
   * Is selected
   * @return whether or not element is selected
   */

  public boolean isSelected()
  {
    return (this==theSelectedElement);
  }

 /**
   * Get font
   * @return get font of element
   */

  public Font getFont()
  {
    return theFont;
  }

 /**
   * Get label width
   * @return get label width of element
   */

  public int getLabelWidth()
  {
    return theLabelWidth;
  }

 /**
   * Get label height
   * @return get label height of element
   */

  public int getLabelHeight()
  {
    return theLabelHeight;
  }
```

```
    /**
     * Set id
     * @param id id of element
     */

    public void setId(int id) {
      theId = id;
      FontMetrics                                   fm                                =
Toolkit.getDefaultToolkit().getFontMetrics(theFont);
      theIdWidth = fm.stringWidth(id+"");
      theIdHeight = fm.getHeight();
    }

    /**
     * Get id
     * @return id of element
     */

    public int getId() {
      return theId;
    }

    /**
     * Get id width
     */

    public int getIdWidth()
    {
      return theIdWidth;
    }

    /**
     * Get id height
     */

    public int getIdHeight()
    {
      return theIdHeight;
    }

    /**
     * Set image icon
     */

    public void setImageIcon(ImageIcon icon, int x, int y)
    {
      theImageIcon = icon;
      theXIcon = x;
      theYIcon = y;
    }

    /**
     * Get image icon
     */

    public ImageIcon getImageIcon()
    {
      return theImageIcon;
    }
```

```
    /**
      * Get x position of icon
      */

    public int getXIcon()
    {
      return theXIcon;
    }

    /**
      * Get y position of icon
      */

    public int getYIcon()
    {
      return theYIcon;
    }

    /**
      * Always associate every wrapper element with the
      * wrapper model it belongs to.
      * Set associated wrapper model.
      */

    public void setWrapperModel(WrapperModel model) {
      theWrapperModel = model;
    }

    /**
      * Get associated wrapper model
      */

    public WrapperModel getWrapperModel() {
      return theWrapperModel;
    }

    //******************** PRIVATE DATA *************************

    // The size of this element.
    public double theSize = 10;
    // The minimum size of this element.
    protected double theMinimumSize = 5;

    // The label associated with this element.
    protected String theLabel = "";
    // The width of the label
    protected int theLabelWidth = 0;
    // The height of the label
    protected int theLabelHeight = 0;

    // The id of this element.
    protected int theId = 0;
    // The width of the id
    protected int theIdWidth = 0;
    // The height of the id
    protected int theIdHeight = 0;
    // Current id (static counter)
    protected static int theCurrentId = 0;
```

```
    // Image Icon to display
    protected ImageIcon theImageIcon = null;
    // X position of image icon
    protected int theXIcon = 0;
    // Y position of image icon
    protected int theYIcon = 0;

    // The font associated with this element.
    protected Font theFont = new Font("Arial",Font.PLAIN,14);

    // Color of this element
    protected Color theColor = Color.black;
    // The selected color of this element
    protected Color theSelectedColor = Color.red;

    // Selected element
    protected static WrapperElement theSelectedElement = null;

    // The wrapper model
    protected WrapperModel theWrapperModel = null;
}
```

APPENDIX C

```
package junglee.vdbms.wdk.visual;
import junglee.vdbms.wdk.interpreter.*;
import junglee.vdbms.wdk.interpreter.Environment;
import java.lang.reflect.*;
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import com.sun.java.swing.*;
import junglee.vdbms.wdk.util.*;

/** An Operation is the basic unit of processing for Visual WDK
 * wrappers.  Each Operation has an <tt>call</tt> method which does
 * the real work, and can call other Operations as
 * needed. Operations use the <tt>setXXX</tt> and <tt>getXXX</tt>
 * naming convention for properties that are set at design time.
 *  <p> As an Operation developer, you should <b>never do
op.call(state)</b>.
 * Instead, you must always use the static method call(from, op,
state).
 * The static method takes care of debugging output and visual trace
 * display; it catches exceptions. Every call should return
 * an Environment, <b>never return null</b> from a call method. **/

/* Stephan Erickson: Added a number of functions & merged the
 * Operation with the GraphModel vertex class. */

public abstract class Operation
  extends WrapperElement
{
  /** Flow of control works by invoking an operation, that is, calling
   * the call method of the operation, passing in any necessary
   * parameters.  This can be started when the user hits the "Run"
   * button (or maybe clicks on the root Operation). The call method
   * runs, and in most cases will call the call method of one of the
   * Operations it is wired up to, and perhaps call repeatedly, in a
   * loop. By default, this call should have the semantics of a normal
   * method call: it waits for the call to complete, and then returns
   * control to the parent call method. We could have the option of
   * running multiple invocations in parallel, but mostly I think that
   * would add to confusion, so let's leave it out (but not forget
   * about it) for the first version. So we have a model where we
   * start at the top, and call methods depth-first as we traverse
   * the web site, and simultaneously work through the wrapper. **/

  /**
    * Constructor
    */

  public Operation()
  {
    theCurrentId++;
    theId = theCurrentId;
    theSize = 15;
    name = shortName(this.getClass().getName());
  }
```

```
    /********************** CALL METHODS ************************/

    /**
     * Every operation must support an implementation of this method
     * @param state the start runtime state
     * @return state the end runtime state
     */

  protected abstract Environment call(Environment state) throws
Throwable;

    /**
     * This static method is what you should call, (within the body of
     * a non-static Operation.call method) to call another operation.
     * That is, do <tt>call(this, getOp(), state)</tt>, not
     * <tt>getOp().call(state)</tt>. The difference is that this static
     * method will gracefully handle a null operation, and it also
     * keeps track of trace information and updates the visual display
     * (if there is one).
     */

  public static Environment call(Operation from, Operation op,
                                 Environment state) throws Throwable {
    if (op == null || from.callsDisabled) return state;
    signal("call", null, state, from, op);
    Environment result = state;
    try {
      result = op.call(state); // <== Real work here
    } catch (Exception e) {
      // Deal with recovering from exception here. ???
      throw e;
    }
    signal("return", null, result, op, from);
    return result;
  }

    /******************        LINKING        OPERATIONS        TOGETHER
**********************/

    /**
     * Every operation must provide the allowable link names via this
function
     * @return list of allowable operation to operation link names
     */

  public abstract String[] getLinkNames();

    /**
     * This function is provided to make access to the link names
easier
     * @return vector of operation link names
     */

  public Vector getLinkNamesVector() {
    String[] linkNames = getLinkNames();
    Vector rval = new Vector();
    for (int i=0; i<linkNames.length; i++) {
      rval.addElement(linkNames[i]);
    }
```

```
    return rval;
  }

  /**
   * Use this method to access the destination Operation of a link.
Links are
   * no longer stored in the operations themselves, but in the
wrapper model.
   * This allows us to avoid writing logic to keep two wrapper
   * representations in sync.
   */

  public Operation getOperation(String linkName)
  {
    Operation op = theWrapperModel.getOperation(this, linkName);
    return op;
  }

   /************************ SIGNAL EVENTS ************************/

  /**
   * Tell any listeners that this event occurred.  Listeners register
   * via WrapperModel.getWrapper().addWrapperListener(cl).
   */

  public static void signal(String eventType,  Object data,
                            Environment state, Operation from,
                            Operation to) {
    if (from!=null) {
      WrapperModel wrapper = from.getWrapperModel();
      Vector listeners = wrapper.getWrapperListeners();
      for (int i = 0; i < listeners.size(); i++) {
        ((WrapperListener)listeners.elementAt(i))
        .eventOccurred(eventType, data, state, from, to);
      }
    }
  }

  /****************** COORDINATE OPERATIONS ***********************/

  /**
   * Move operation to absolute coordinates x,y
   * @param x destination x coordinate
   * @param y destination y coordinate
   */

  public void moveTo(int x,int y)
  {
    x0 = x;
    y0 = y;
  }

  /**
   * Move operation relative to coordinates x0,y0
   * @param deltaX relative x coordinate
   * @param deltaY relative y coordinate
   */

  public void moveBy(int deltaX,int deltaY) {
```

```
    x0+=deltaX;
    y0+=deltaY;
  }

  /**
    * Scale size of operation using x as multiplier
    * @param x scale multiplier
    */

  public void scale(double x) {
    if (theSize*x>theMinimumSize) {
      x0 *= x;
      y0 *= x;
    }
  }

  /**
    * Return distance from x,y to operation
    * @param x the x coordinate
    * @param y the y coordinate
    */

  public int distanceFrom(int x,int y) {
    return (int) Math.sqrt((x-x0)*(x-x0)+(y-y0)*(y-y0));
  }

  /********************** FOLLOWING LINKS *****************/

  /**
    * Follow link from url. Specify post query if POST
    * is required, otherwise specify null for the
    * postQuery parameter
    * @param url in string form
    * @param post query in string form (EG: a=34&b=this is a test&c=0)
    */

  public String followLink(String url, String postQuery)
  {
    URL inUrl = null;
    String text = "";
    try {
      inUrl = new URL(url);
      text = netClient.getText(inUrl);
    } catch(Exception e) {
      e.printStackTrace();
      // SFE change
    }
    return text;
  }

  public URL convertUrl(String url)
  {
    URL newUrl = null;
    try {
      newUrl = new URL(url);
    } catch(Exception ex) {
      ex.printStackTrace(); // SFE change
    }
    return newUrl;
```

```
  }

  /****************** PROPERTY SETTER/GETTERS ****************/

  /**
    * The name of an Operation is a short label for use by the human,
    * and for debugging output. It defaults to the unqualified class
name,
    * but can be set to anything you want.
    */

  public String getName() { return name; }
  public void setName(String name) { this.name = name; }

  /**
    * The comment of an Operation can be a longish comment block
    * describing what this operation is used for in this wrapper.
    * There will also be Help documentation that says what each
Operation
    * class is for, and how to use it.  But that comes later.
    */

  public String getComment() { return comment; }
  public void setComment(String desc) { this.comment = desc; }

  /**
    * The coordinates of the operation
    */

  public void setX(int x) { x0 = (double)x; }
  public int getX() { return (int)x0; }
  public void setY(int y) { y0 = (double)y; }
  public int getY() { return (int)y0; }

  /**
    * Given a qualified class name as argument, return the unqualified
part.
    * That is, the part after the last ".".  Also remove "Operation"
if it is
    * at the end.
    */

  public static String shortName(String className) {
    if (className.endsWith("Operation"))
          className = className.substring(0, className.length()-9);
    int dot = className.lastIndexOf('.');
    if (dot == -1) return className;
    else return className.substring(dot+1);
  }

  /**
    * Return the operation's name.
    */

  public String toString() { return getName(); }


  /****************** DATA ************************/
```

```
  /** Name of operation **/
  String name = "";
  /** Comment associated with operation **/
  String comment = "";

  /** Should we print information for debugging WrapperBuilder? This
is
   * for a developer debugging WrapperBuilder; not a user debugging
   * a wrapper. **/
  static boolean debugging = true;

  // The net client static object used to support followLink
  static Web netClient = new Web();

  /** Is this operation a breakpoint? **/
  public boolean isBreakpoint = false;
 /** Are calls disabled? **/
  public boolean callsDisabled = false;

  /* x coordinate of operation center */
  public double x0;
  /* y coordinate of operation center */
  public double y0;
}
```

APPENDIX D

```
package junglee.vdbms.wdk.operations;
import junglee.vdbms.wdk.visual.*;
import junglee.vdbms.wdk.visualgui.*;
import junglee.vdbms.wdk.interpreter.*;
import junglee.vdbms.wdk.interpreter.Environment;
import junglee.vdbms.wdk.util.*;
import java.util.*;
import java.net.*;

/**
  * Match Operation
  */

public class MatchOperation extends Operation {

  /**
    * Constructor
    */

  public MatchOperation() {}

  /**
    * Iterate over text using the match expression entered
    * in the Match property.
    *
    * Additional options are:
    *   - Read initial text from URL (URL property)
    *   - Follow URLs after match (FollowLinks property)
    */

  public final Environment call(Environment state)
    throws Throwable
  {
    // Create new copy of state
    /*
    Environment newState = new Environment();
    state.transfer(newState);
    */

    // Temporarily make mutable to debug memory usage SFE change
    Environment newState = state;
    Interpreter interpreter = new Interpreter(newState);

    // Fetch url, input text, start position, end position
    String parentUrl = state.evaluateVariable(Environment.URL);
    String input = state.evaluateVariable(Environment.TEXT);
    String startStr = state.evaluateVariable(Environment.START);
    String endStr = state.evaluateVariable(Environment.END);
    int startInt = -1;
    int endInt = -1;

    // Obtain start position
    if (startStr!=null) {
      Integer startInteger = new Integer(startStr);
      startInt = startInteger.intValue();
    }
    // Obtain end position
```

```
        if (endStr!=null) {
          Integer endInteger = new Integer(endStr);
          endInt = endInteger.intValue();
        }
        input = input.substring(startInt, endInt);

        // If start url specified
        if (!startUrl.equals("")) {
          input = followLink(startUrl, null);
        }

        // Initialize input & match expression
        newState.setVariable(Environment.TEXT, input, false);

        Vector vars = interpreter.getVariables(match);

        // Iteratively match
        for (;;) {
          Vector expressions = interpreter.parseExpressionSequence(match);
          interpreter.evaluateExpressionSequence(expressions);
          if (!interpreter.hasMoreInput()) break;

          // Call link for each variable
          Enumeration e = vars.elements();
          for (;e.hasMoreElements();) {
            String variable = (String)e.nextElement();
            String value = newState.evaluateVariable(variable);

            // If followLinks true, follow URLs
            if (followLinks.booleanValue()) {
              TextParser parser = new TextParser(value);
              // Iterate through each url
              for (;parser.hasMoreUrls();) {
                String url = parser.nextUrl(parentUrl);
                if (url==null) continue;
                // Eliminate duplicates
                if (theTable.get(url)==null) {
                  theTable.put(url, "");
                } else {
                  continue;
                }

                String newPage = followLink(url, null);
                newState.setVariable(Environment.TEXT, newPage, false);
                newState.setVariable(Environment.URL, url, false);
                newState.setVariable(Environment.START, ""+0, false);
                newState.setVariable(Environment.END,  ""+newPage.length(),
false);
                /*  Environment  tmpState  =  */  newState  =  call(this,
getOperation(variable), newState);
                /* tmpState.transfer(state); SFE change */
              }
            } else {
              // If followLinks false, do not follow URLs
              int start = 0;
              int end = 0;
              try {
                start = interpreter.getMatchStart(variable);
                end = interpreter.getMatchEnd(variable);
```

```
            } catch (Exception ex) {
              // No selection on failure
            }
            newState.setVariable(Environment.TEXT, input, false);
            newState.setVariable(Environment.START, ""+start, false);
            newState.setVariable(Environment.END, ""+end, false);
            /*  Environment  tmpState  =  */  newState  =  call(this,
getOperation(variable), newState);
            /* tmpState.transfer(state); SFE change */
          }
        }
      }
      // Emit rows if necessary
      if (!emitRows.equals("")) {
        interpreter.evaluateExpression("emit-rows('"+emitRows+"')", 0);
      }
      return state;
    }

    /** Return set of link names used by this operation **/
    public String[] getLinkNames()
    {
      Environment env = new Environment();
      Interpreter interpreter = new Interpreter(env);
      Vector linkVector = interpreter.getVariables(match);
      String[] linkArray = new String[linkVector.size()];
      linkVector.copyInto((String[])linkArray);
      return linkArray;
    }

    ///////////////// Methods to define Properties ///////////////

    String match = "";
    public String getMatch() { return match; }
    public void setMatch(String match) { this.match = match; }

    String startUrl = "";
    public String getStartUrl() { return startUrl; }
    public void setStartUrl(String url) { this.startUrl = url; }

    Boolean followLinks = new Boolean(false);
    public Boolean getFollowLinks() { return followLinks; }
    public void setFollowLinks(Boolean followLinks) { this.followLinks =
followLinks; }

    String emitRows = new String("");
    public String getEmitRows() { return emitRows; }
    public void setEmitRows(String emitRows) { this.emitRows = emitRows;
}

    // Eliminate duplicates
    public static Hashtable theTable;
}
```

APPENDIX E

```
package junglee.vdbms.wdk.visual;
import java.awt.*;
import java.io.*;

/**
  * <b> Link Class </b>
  *
  * <p>This class is used to represent links in a wrapper model.
  * A link is identified as a directed line connecting two operations.
  * A link is associated with: <br>
  * <ul>
  *   <li> A start operation </li>
  *   <li> An end operation </li>
  * </ul>
  * <p>Copyright (c) 1997, Junglee Corp.
  * @title Link
  * @author Stephan Erickson
  * @version 1.0
  */

public class Link
  extends WrapperElement
  implements Serializable
{

  //******************* MODEL OPERATIONS ********************

  public Link()
  {
    super();
  }

  /**
    * Constructor.
    * Provide start and end operations. Default constructor
    * returns a directed link with arrow pointing to end operation.
    * Does nothing if either operation is null
    * @param start the start operation
    * @param end the end operation
    */

  public Link(Operation start, Operation end)
  {
    theStartOperation = start;
    theStartOperationId = start.getId();
    theEndOperation = end;
    theEndOperationId = end.getId();
  }

  /**
    * Return start operation of link
    * @return Start Operation
    */

  public Operation getStartOperation()
  {
    return theStartOperation;
```

```
}

/**
  * Set start operation of link
  * @param Start Operation
  */

public void setStartOperation(Operation start)
{
  theStartOperation = start;
  theStartOperationId = start.getId();
}

 /**
  * Set end operation of link
  * @param end Operation
  */

public void setEndOperation(Operation end)
{
  theEndOperation = end;
  theEndOperationId = end.getId();
}

/**
  * Return end operation of link
  * @return End Operation
  */

public Operation getEndOperation()
{
  return theEndOperation;
}

/**
  * Set start operation id of link
  */

public void setStartOperationId(int id)
{
  theStartOperationId = id;
}


/**
  * Return start operation id of link
  * @return Start Operation Id
  */

public int getStartOperationId()
{
  return theStartOperationId;
}

/**
  * Set end operation id of link
  */

public void setEndOperationId(int id)
```

```
  {
    theEndOperationId = id;
  }

  /**
   * Return end operation id of link
   * @return End Operation Id
   */

  public int getEndOperationId()
  {
    return theEndOperationId;
  }

  //******************** GRAPHICS OPERATIONS ********************

  /**
   * Distance from link
   * @param x the x coordinate
   * @param y the y coordinate
   * @return distance from link
   */

  public double distanceFrom(double x, double y)
  {
    double length =
      Math.sqrt((theEndOperation.x0-theStartOperation.x0)*
                (theEndOperation.x0-theStartOperation.x0)+
                (theEndOperation.y0-theStartOperation.y0)*
                (theEndOperation.y0-theStartOperation.y0));
    double cos = (theEndOperation.x0-theStartOperation.x0)/length;
    double sin = (theEndOperation.y0-theStartOperation.y0)/length;
    double distance = Math.abs((theStartOperation.x0-x)*sin+
                      (y-theStartOperation.y0)*cos);
    return distance;
  }

  //*********************** PRIVATE VARIABLES **********************

  /* The start operation */
  private Operation theStartOperation;

  /* The end operation */
  private Operation theEndOperation;

  /* The start operation id */
  private int theStartOperationId;

  /* The end operation id */
  private int theEndOperationId;
}
```